Programación

Unidad 9
Ejercicio Guiado
Zoo

Jesús Alberto Martínez
Versión 0.1

Ejercicio Guiado. Zoo

1 Primera parte. Enunciado

Debemos implementar un programa que llevar un control de los animales de un Zoo.

Un Animal tiene un nombre y un número de estancia. Se considera que dos animales son iguales si tienen el mismo nombre, y una vez establecido el nombre, no se puede modificar.

El zoo está formado por un conjunto de animales, donde se permiten realizar las siguientes operaciones:

- boolean añadirAnimal(Animal a). Añade el animal al Zoo, No se permiten que dupliquemos el nombre del animal en otra instancia. Indicará si el animal se ha introducido en el Zoo o no.
- boolean eliminarAnimal(String nombre) Elimina el animal del Zoo con ese nombre. Indicará si se ha podido eliminar.
- int buscarAnimal(String nombre). busca un animal por su nombre, devuelve su posición en el Zoo.
- void listarAnimales(). Muestra por pantalla la lista de animales junto con su estancia del Zoo.

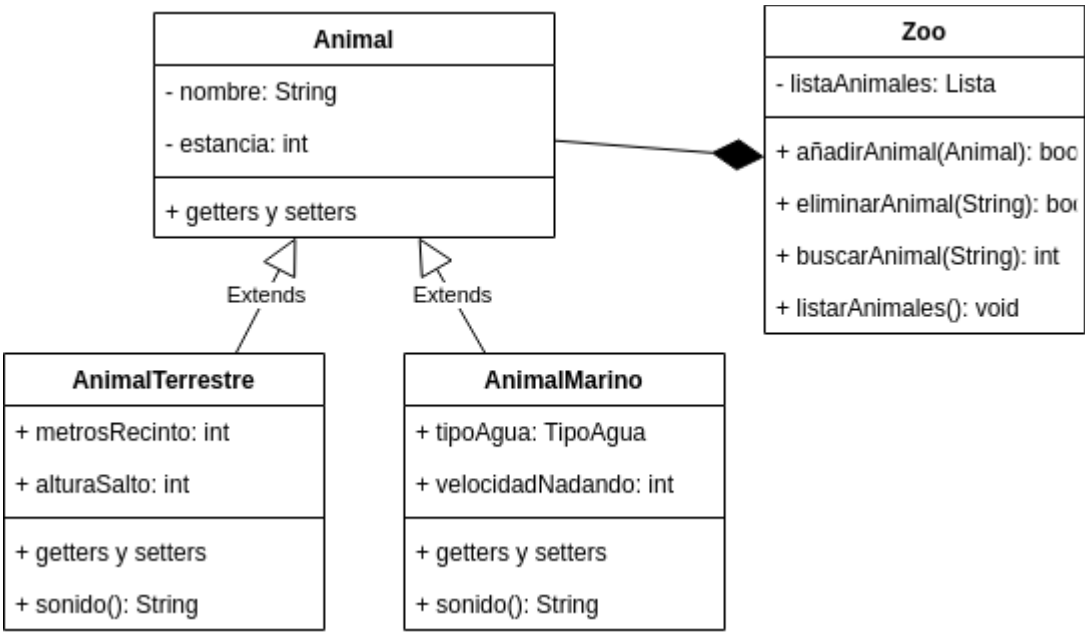
En el Zoo distinguimos entre animales, AnimalTerrestre y AnimalMarino, para los animales terrestres guardamos, además del nombre y el número de estancia, los metros de recinto y cuanto puede saltar, y para los animales marinos guardamos, además del nombre y el número de estancia, si es de agua salada o dulce, y su velocidad nadando. Un animal siempre será de uno de estos tipos.

Además todos los tipos de animales tiene un método ejercitar() que hará que los animales hagan ejercicio, cada uno lo hará de una forma distinta.

Implementa las clases Animal, AnimalTerrestre, AnimalMarino y Zoo. Implementa también la clase ProgramaZoo con una función main para realizar pruebas.

1.1 El diagrama de clases

Vamos a crear un diagrama de clases según el enunciado. Leemos paso a paso y vamos anotando las requerimientos.



1.2 Clase Animal

Comencemos implementando la clase Animal

> Un Animal tiene un nombre y un número de estancia.

```
public class Animal {

   private String nombre;
   private int estancia;
```

No nos indica nada, pero debemos seguir uno de los principios de la POO, la encapsulación, y definir los atributos con el nivel de visibilidad más restrictivo posible si no nos indican lo contrario.

De la misma forma, no nos dice nada sobre la creación de los objetos, pero siempre en la definición de una clase debemos pensar en la creación de los objetos de esa clase y proporcionar un constructor que al crear los objetos queden bien definidos. Lo normal es tener siempre un constructor donde se le especifiquen todos los atributos propios del objeto.

```
   public Animal(String nombre, int estancia) {
         super();
         this.nombre = nombre;
         this.estancia = estancia;
   }
```

Como hemos establecido los atributos como privados, necesitaremos métodos para poder obtener y/o modificar sus valores, los getters y setters.

```
public String getNombre() {
	return nombre;
}

public void setNombre(String nombre) {
	this.nombre=nombre;
}

public int getEstancia() {
	return estancia;
}

public void setEstancia(int estancia) {
	this.estancia = estancia;
}
```

> Se considera que dos animales son iguales si tienen el mismo nombre

Esta frase nos indica cómo considerar que dos objetos de la clase Animal se consideran iguales. Todos los objetos tiene definido un método, **equals**, que utiliza Java en muchos métodos internos para comprobar si dos objetos son iguales. En casi todos los objetos de Java están ya definidos, pero en los objetos que creamos nosotros debemos definirlo sobreescribiendo el método equals, de echo, los entornos de desarrollo nos facilitarán esta implementación.

Para sobreescribir un método de una clase padre, el método debe tener la misma firma,debe ser exactamente igual. La firma del método equals es:

```
public boolean equals(Object obj)
```

Lo sobreescribimos

```
@Override
public boolean equals(Object obj) {
	if (this == obj)
		return true;
	if (!(obj instanceof Animal))
		return false;
	return this.nombre.equals(((Animal)obj).nombre);
}
```

Estamos indicando que un objeto Animal será igual a otro si, el objeto es una instancia de Animal, eso será válido para los objetos Animal y cualquier subclase de ella, también se podría hacer con getClass, pero entonces diferenciaría entre los objetos de las subclases, y una vez que estamos seguros que es un

Animal, podemos hacer un moldeado del objeto (cast), y comprobar si sus nombres son iguales, al final devolveremos el resultado de esa comprobación

> , y una vez establecido el nombre, no se puede modificar.

Para indicar que el valor de un atributo no se puede modificar una vez establecido debemos poner la palabra clave **final** en la definición del atributo.

```
    private final String nombre;
```

Esto tiene otras consecuencias, nos forzaría a tener un constructor donde se le pasara el nombre si no lo tuviéramos, además nos dará error en todos los métodos que intentaran modificar el valor de ese atributo.

1.3 Clase Zoo

> El zoo está formado por un conjunto de animales, donde se permiten realizar las siguientes operaciones:

Nos indica que el contenido de la clase van a ser animales, en un número desconocido. Por lo que vamos a usar una estructura de datos que pueda crecer de forma dinámica, y de momento, la única que conocemos es ArrayList

```
public class Zoo {

    private ArrayList<Animal> listaAnimales;
```

El ArrayList es de Animal, por polimorfismo, podrá contener objetos de la clase Animal o de cualquier objeto que sea un Animal, es decir, que herede de esta clase. El atributo está definido, pero no está inicializado. Nos creamos un constructor para ello.

```
    public Zoo() {
          listaAnimales=new ArrayList<>();
    }
```

Se puede inicializar el atributo en la definición y no en el constructor. Sólo debemos tener en cuenta, que si tenemos inicializaciones en las definiciones, y después tenemos constructor, primero se realiza lo que está en la definición, y después se realizará lo que está en el constructor.

Vamos a ver los métodos que nos piden. En principio no necesitamos getter o setter, ya que la gestión de la lista la queremos realizar con sus métodos.

- boolean añadirAnimal(Animal a). Añade el animal al Zoo, No se permiten que dupliquemos el nombre del animal en otra instancia. Indicará si el animal se ha introducido en el Zoo o no.

Este método añade animales a la lista del Zoo, le pasamos como parámetro un animal, por lo que quedaría añadido simplemente con un

```
            listaAnimales.add(a);
```

Pero fuerza a que no haya animales repetidos en otras estancias por lo que tenemos que comprobar primero si ya existe el animal. Podemos hacerlo de varias formas, recorriendo los animales de la lista y comprobando su nombre

```
        for (int i = 0; i < listaAnimales.size(); i++) {
            if (listaAnimales.get(i).getNombre().equals(animal.nombre))
```

Pero ya que tenemos implementado un método que nos dice si un Animal es iguala otro, podríamos usar:

```
        for (int i = 0; i < listaAnimales.size(); i++) {
            if (listaAnimales.get(i).equals(animal))
```

Además tenemos otras formas, como hemos dicho Java usa internamente el método equals en muchos métodos. Por ejemplo el método indexOf, que nos devuelve el primer índice de un objeto dentro de un array o -1 si no lo encuentra, usa internamente equals para saber si un objeto es igual a otro, por lo que podríamos usar:

```
        if (listaAnimales.indexOf(animal)!=-1)
```

También el método contains que nos dice si un array tiene o no un objeto dentro usa equals para comprobarlo. Así vemos a dejar el método al final:

```
    public boolean añadirAnimal(Animal a) {
        if (listaAnimales.contains(a))
            return false;
        else
            listaAnimales.add(a);
        return true;
    }
```

- boolean eliminarAnimal(String nombre) Elimina el animal del Zoo con ese nombre. Indicará si se ha podido eliminar.

Aquí buscamos el animal y lo borramos si lo encontramos. Después me piden un método para buscar un animal, ¿por qué no usarlo si me va a decir si un animal está o no es la lista?

```
public boolean eliminarAnimal(String nombre) {
	int indice=buscarAnimal(nombre);
	if (indice!=-1) {
		listaAnimales.remove(indice);
		return true;
	}
	else
		return false;
}
```

- int buscarAnimal(String nombre). busca un animal por su nombre, devuelve su posición en el Zoo.

Como vimos en el método de añadir, tenemos muchas alternativas para buscar un animal, todas ellas válidas, y en la primera de ellas usamos una búsqueda por nombre.

```
int buscarAnimal(String nombre) {
	for (int i = 0; i < listaAnimales.size(); i++) {
		if (listaAnimales.get(i).getNombre().equals(nombre))
			return i;
	}
	return -1;
}
```

Como siempre podemos hacerlo de otra forma

```
public int buscarAnimal(String nombre) {
	return listaAnimales.indexOf(new Animal(nombre,0));
}
```

Recordemos, va a buscar objetos que son iguales... y son iguales si tienen el mismo nombre.

- void listarAnimales(). Muestra por pantalla la lista de animales junto con su estancia del Zoo.

La forma más sencilla de realizar esto es recorrer la lista de animales y mostrar su información usando el método toString, de nuevo un método que tienen todos los objetos, heredado de la superclase Object y que tenemos que sobreescribir para mostrar la información exacta que queremos mostrar. Por ejemplo:

```
    @Override
    public String toString() {
          return "Nombre=" + nombre + ", Estancia=" + estancia;
    }
```

Y la implementación del método

```
    public void listaAnimales() {
          System.out.println("Animales en el Zoo");
          for (Animal a: listaAnimales) {
                System.out.println(a);
          }
    }
```

1.4 Herencias animales

> En el Zoo distinguimos entre animales, AnimalTerrestre y AnimalMarino, para los animales terrestres guardamos, además del nombre y el número de estancia, los metros de recinto y cuanto puede saltar,

Tenemos aquí una relación de herencia, tanto AnimalTerrestre como AnimalMarino cumplen que **son** Animal, son **especializaciones** de Animal. Heredamos de Animal, pudiendo así usar sus métodos y teniendo sus atributos, aunque sean privados. Añadimos los atributos propios de las nuevas clases

```
public class AnimalTerrestre extends Animal {

    private int recinto;
    private int longitudSalto;

Tenemos que crear además un constructor adecuada a nuestros nuevos objetos
    public AnimalTerrestre(String nombre, int estancia, int recinto, int
longitudSalto) {
          super(nombre, estancia);
          this.recinto = recinto;
          this.longitudSalto = longitudSalto;
    }
```

La primera línea del constructor es la llamada al constructor de la clase padre con los parámetros adecuados. Siempre se llama al constructor de la clase padre, si no ponemos nada, se llama al constructor por defecto, el que no lleva parámetros, aunque no lo pongamos. En este caso, no existe constructor por defecto porque hemos definido un constructor distinto, y no hemos definido el constructor por defecto. Si no ponemos la primera línea nos dará error.

Establecemos los métodos para acceder y modificar los atributos por si nos hicieran falta.

```
    public int getRecinto() {
          return recinto;
    }


    public void setRecinto(int recinto) {
          this.recinto = recinto;
    }


    public int getLongitudSalto() {
          return longitudSalto;
    }


    public void setLongitudSalto(int longitudSalto) {
          this.longitudSalto = longitudSalto;
    }
```

> y para los animales marinos guardamos, además del nombre y el número de estancia, si es de agua salada o dulce, y su velocidad nadando.

Con los animales marinos realizamos la misma operación

```
public class AnimalMarino extends Animal {

    private TipoAgua tipoAgua;
    private int velocidad;
```

con su constructor adecuado

```
    public AnimalMarino(String nombre, int estancia, TipoAgua tipoAgua, int
velocidad) {
          super(nombre, estancia);
          this.tipoAgua = tipoAgua;
          this.velocidad = velocidad;
    }
```

Y sus métodos

```
    public TipoAgua getTipoAgua() {
          return tipoAgua;
    }

    public void setTipoAgua(TipoAgua tipoAgua) {
          this.tipoAgua = tipoAgua;
    }
```

```
    public int getVelocidad() {
          return velocidad;
    }

    public void setVelocidad(int velocidad) {
          this.velocidad = velocidad;
    }
```

1.5 Enumeraciones

Si nos fijamos, el tipo de agua del AnimalMarino está definida como TipoAgua. Podriamos haber optado por poner simplemente una cadena de texto, pero para evitar errores en los datos, se usan las **enumeraciones**.

Es como crear un nuevo tipo de datos, donde se establecen los valores que puede contener ese tipo de datos. Por ejemplo esta enumeración está definida así:

```
public enum TipoAgua {

    SALADA, DULCE
}
```

Para usarla tenemos que usar este tipo de datos TipoAgua, con sus valores, TipoAgua.SALADA o TipoAgua.DULCE , cualquier otro valor no será válido.

1.6 Animales abstractos

> Un animal siempre será de uno de estos tipos.

Esta frase es muy particular, nos dice realmente que no vamos a usar nunca la clase Animal, no vamos a tener objetos de esta clase. Java nos proporciona un mecanismo para poder especificar esto, consiste en definir la clase Animal como abstracta, con esto no podremos crear objetos, la clase está pensada para que otras clases hereden de ella.

```
public abstract class Animal {
```

Esta modificación conlleva que tengamos que modificar el método de buscarAnimal de Zoo, ya que no podemos crear objetos Animal, y tendremos que realizar la búsqueda de otra manera.

> Además todos los tipos de animales tiene un método ejercitar() que hará que los animales hagan ejercicio, cada uno lo hará de una forma distinta.

Esta frase del enunciado suponen más cambios, indica un nuevo método en las clases que heredan de Animal, pero no queda claro como se hará exactamente, cada uno hará una cosa distinta. El mecanismo que nos proporciona la POO para especificar esto es definiendo un método como abstracto en Animal, y lo definimos indicando su firma, pero sin implementar.

```
public abstract void ejercicio();
```

Si ponemos un método abstracto en una clase, forzosamente la clase tiene que ser abstracta. Y al tener un método abstracto, es obligatorio implementar ese método en las subclases (a no ser que sean también abstractas), por lo que ahora tenemos errores en las clases AnimalTerrestre y AnimalMarino que solucionamos implementando el método abstracto de la clase padre.

Como es sólo ilustrativo, vamos a implementar los métodos de esta forma:

```
@Override
public void ejercicio() {
      System.out.println("El animal marino hace ejercicio");
}
```

1.7 ProgramaZoo (primera parte)

Creamos un programa simple para probar los constructores de Animal y los métodos de Zoo. Fijaros como una variable de tipo Animal puede contener cualquier objeto de subclases.

```
Animal a1=new AnimalTerrestre("Macaco",1,12,8);
AnimalTerrestre a2=new AnimalTerrestre("Pavo Real",2,10,5);
Animal a3=new AnimalMarino("Nutria",21,TipoAgua.DULCE,20);
Animal a4=new AnimalTerrestre("Macaco",22,6,12);
Animal a5=new AnimalMarino("Macaco",21,TipoAgua.DULCE,20);

Zoo zoo=new Zoo();
System.out.println(zoo.añadirAnimal(a1));
System.out.println(zoo.añadirAnimal(a2));
System.out.println(zoo.añadirAnimal(a3));
System.out.println(zoo.añadirAnimal(a4));
System.out.println(zoo.añadirAnimal(a5));
System.out.println(zoo.añadirAnimal(new AnimalMarino("Delfín",21,TipoAgua.SALADA,40)));

zoo.listaAnimales();

System.out.println("Índice del mono:"+zoo.buscarAnimal("Macaco"));
zoo.eliminarAnimal("Macaco");

zoo.listaAnimales();
```

2 Segunda Parte. Enunciado

Por otro lado tenemos una clase llamada TriAtleta, con un nombre, una velocidad corriendo, una velocidad nadando y una velocidad en bicicleta, expresadas las tres en kilometros por hora. Encapsula sus atributos, crea un constructor y los métodos de acceso y modificación adecuados.

Queremos realizar una comparación entre animales del Zoo y TriAtletas, para ello vamos a desarrollar dos interfaces, Nadador y Saltador. En nadador tenemos un método:

- int nadar(int metros). Se le pasará una cantidad de metros y nos devolverá el número de segundos que tarda en nadar la distancia.

En la interfaz Saltador tendremos el siguiente método:

- boolean saltarAltura(int centimetros). Nos indicará si puede o no puede saltar la altura que se le pasa como parámetro.

Triatleta implementará ambas interfaces, los animales terrestres implementarán la interfaz Saltador y los marinos la interfaz Nadador.

En el programa de pruebas, crea un ArrayList de Nadadores que contenga tanto a TriAtletas como AnimalesMarinos y muestra lo que tardan en recorrer 500 metros.

2.1 El triatleta

> Por otro lado tenemos una clase llamada TriAtleta, con un nombre, una velocidad corriendo, una velocidad nadando y una velocidad en bicicleta, expresadas las tres en kilometros por hora. Encapsula sus atributos, crea un constructor y los métodos de acceso y modificación adecuados.

Creamos la clase con los atributos y métodos

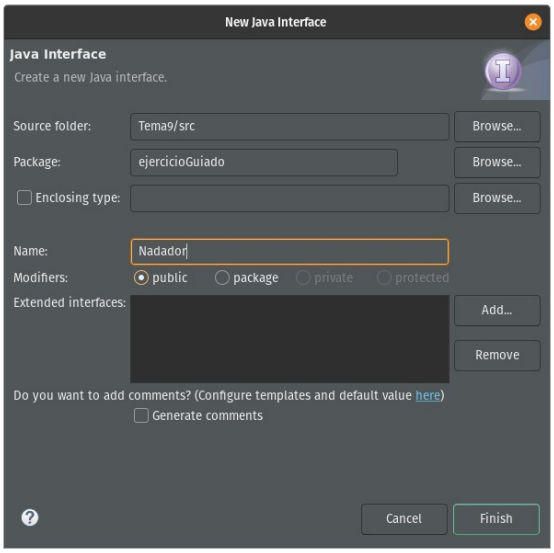
```
public class TriAtleta {

    private String nombre;
    private int velCorriendo;
    private int velNadando;
    private int velBicicleta;

    public TriAtleta(String nombre, int velCorriendo, int velNadando, int velBicicleta) {
…
```

2.2 Las interfaces

Una interface es una herramienta para la confección de clases. Define una serie de constantes y de métodos, únicamente las define, no las implementa, como pasaba con los métodos abstractos.

Sirven para proporcionar una funcionalidad, la herencia proporciona una relación entre clases, la implementación de una interfaz proporciona una funcionalidad común entre clases.

Para definir una interfaz la realizamos de forma parecida a como realizamos las clases, pero esta vez seleccionamos interfaz,



> Queremos realizar una comparación entre animales del Zoo y TriAtletas, para ello vamos a desarrollar dos interfaces, Nadador y Saltador. En nadador tenemos un método:
> int nadar(int metros). Se le pasará una cantidad de metros y nos devolverá el número de segundos que tarda en nadar la distancia.

Nuestra interfaz Nadador viene definida por un sólo método, así que la definición de la interfaz será:

```
public interface Nadador {
    int nadar(int metros);
}
```

> En la interfaz Saltador tendremos el siguiente método:
> boolean saltarAltura(int centimetros). Nos indicará si puede o no puede saltar la altura que se le pasa como parámetro.

Y la interfaz Saltador

```
public interface Saltador {
    boolean saltar(int centimetros);
}
```

Aunque no se pongan, los métodos se definen como públicos.

Con esta definición indicamos que cualquier clase que quiera implementar la interfaz, debe tener los métodos que en ella se indican, tienen la funcionalidad de la interfaz, en este caso , nadar o saltar.

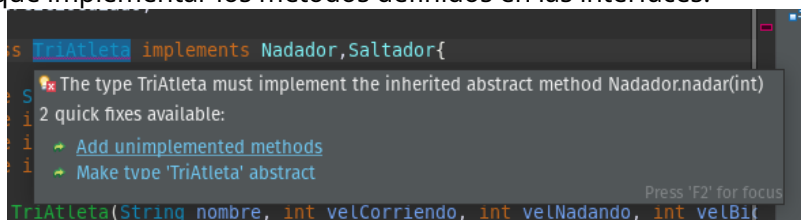
2.3 Implementando las interface

Una clase sólo puede heredar de una clase, pero puede implementar cualquier número de interfaces. Para ello se especifica con la palabra clave **implements** y a continuación la lista de interfaces que implementa, separadas por comas.

Triatleta implementará ambas interfaces,

```
public class TriAtleta implements Nadador,Saltador{
```

Al indicar que TriAtleta implementa las interfaces, automáticamente nos salta un error en la clase, nos indica que tenemos que implementar los métodos definidos en las interfaces.



Generamos los métodos que debemos implementar, y codificamos lo que hacen los métodos. Por ejemplo podría ser esto:

```
    @Override
    public boolean saltar(int centimetros) {
            // Los triatletas no saltan
            return false;
    }

    @Override
    public int nadar(int metros) {
            // Su velocidad es en kilometros por hora.
            // la pasamos a metros por segundo
            double metrosSegundo=this.velNadando / 0.27777778;
            return (int)(metros*metrosSegundo);
    }
```

los animales terrestres implementarán la interfaz Saltador

Nos vamos a la clase e indicamos que queremos implementar la interfaz

```
public class AnimalTerrestre extends Animal implements Saltador{
```

Y ahora tenemos que implementar el método

```
    @Override
    public boolean saltar(int centimetros) {
           return this.longitudSalto>centimetros;
    }
```

> y los marinos la interfaz Nadador.

```
public class AnimalMarino extends Animal implements Nadador {
```

Y codificamos,

```
    @Override
    public int nadar(int metros) {
           double metrosSegundo=this.velocidad / 0.27777778;
           return (int)(metros*metrosSegundo);
    }
```

2.4 Programa Zoo. Segunda parte

Y todo esto ¿para qué?

> En el programa de pruebas, crea un ArrayList de Nadadores que contenga tanto a TriAtletas como AnimalesMarinos y muestra lo que tardan en recorrer 500 metros.

La implementación de las interfaces hace que las clases tengan algo en común, no es herencia múltiple, no compartimos atributos, pero sí métodos. Podemos decir que un TriAtleta y un AnimalMarino tienen algo en común, nadan, son Nadadores, y por **polimorfismo de interfaces** podemos hacer que ambos estén por ejemplo en un mismo ArrayList, y usar los métodos definidos en la interface.

```
           ArrayList<Nadador> nadadores=new ArrayList<>();
           nadadores.add(new AnimalMarino("Nutria",21,TipoAgua.DULCE,20));
           nadadores.add(new TriAtleta("Atleta 1",10,3,30));
           for (Nadador n: nadadores) {
                  System.out.println(n+" tarda en nadar 500 metros "+n.nadar(500)+" segundos, a
una velocidad de "+n.velocidadMS()+" m/s");
```

Para este ejemplo se ha definido en la interface Nadador otro método velocidadMS.